Science of Computer Programming 21 (1993) 191–223
Elsevier

Rankers: a classification of synchronization problems

Ambuj K. Singh *

Department of Computer Science, University of California, Santa Barbara, CA 93106, USA

Mohamed G. Gouda †

Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, USA

Communicated by M. Sintzoff
Received February 1992
Revised April 1993

Abstract

Singh, A.K. and M.G. Gouda, Rankers: a classification of synchronization problems, Science of Computer Programming 21 (1993) 191–223.

The need for synchronization of concurrent computation arises due to various reasons. All such instances where synchronization is required have been abstracted into a few well-known problems: producer–consumer, mutual exclusion, dining philosophers, etc. This paper presents a general mechanism and an accompanying methodology for solving such synchronization problems. The paper starts by recognizing a common property of all solutions to synchronization problems, namely the presence of some form of arbitration. This idea is crystallized by introducing *ranks* as a mechanism for carrying out arbitration, and by introducing *rankers* as a distributed module that computes ranks upon request. Mandatory and optional properties of rankers are specified. Based on this specification, a strict hierarchy of four increasingly powerful rankers is identified. Each ranker in this hierarchy is best suited to solve a particular class of synchronization problems and together the four rankers are sufficient for solving most synchronization problems in the literature. A general methodology for solving synchronization problems using rankers is developed. This methodology is illustrated by solving a number of synchronization problems from the literature.

Correspondence to: A.K. Singh, Department of Computer Science, University of California, Santa Barbara, CA 93106, USA. E-mail: ambuj@cs.ucsb.edu.

*Work supported in part by ONR Grants N00014-86-K-0763 and N00014-87-K-0510 and NSF grant CCR-9008628.

†Work supported in part by ONR Grant N00014-86-K-0763.

1. Introduction

Processes in a distributed system usually need to coordinate or synchronize their actions. The need for such synchronization arises due to a number of reasons, some of which are as follows:

- *Bounded resources.* The need for synchronization may arise because the number of resources is less than the number of processes wanting to use them. Classic resource allocation problems such as mutual exclusion [13], dining philosophers [14], readers–writers [11], and CSP rendezvous [6] fall in this class.
- *Order preservation.* It is sometimes necessary to preserve the order of actions across processes. For example, in the producer–consumer problem [5] the order of consumption of resources should be same as the order of production; similarly, in the FCFS doorway problem [22] the order in which processes exit the doorway should reflect the order in which they entered it; and in the processor renaming problem [3] the initial order of names should be preserved in the new ordering.
- *Concurrency control.* The semantics (correctness) of computations is sometimes defined only for sequential (i.e., single-process) executions, and then synchronization is needed to ensure that a concurrent execution is equivalent to some sequential execution. The problems of ensuring serializability in databases [17] and linearizability in concurrent objects [19] and atomic registers [4,7,27,30,32,38] fall in this class.

We refer to the classes of problems mentioned above broadly as synchronization problems. Synchronization problems have been studied and solved by numerous researchers. Most of these solutions, however, are specific to the particular computing architecture on which the solutions are to be implemented. For example, consider the mutual exclusion problem. It has been solved for the shared-memory architecture using schemes as varied as semaphores [12] (weak and strong, binary and n-ary), tokens [8], and tickets [22], and for the message-passing architecture using time-stamps [24,36], quorums [1], tree-based arbitration [34], and acyclic graph-based arbitration [9]. In fact new solutions, each specific to a particular computing architecture, are being continually designed for these problems.

Given the multitude of synchronization problems and their solutions, one is led to search for a unifying framework—a framework in which these problems can be stated and solved modularly at a high level and through which existing solutions to the problems on various architectures can be compared. There are three principal benefits of such a framework. First, the framework can be used to explain and formally state the relationship between these problems. This will clarify some of the vague relationships existing in the synchronization folklore. Second, the framework allows us to focus on the intrinsic requirements

of a given problem. These requirements are independent of any architectural considerations and should be met by all solutions to the given problem. The identification of these requirements allows us to classify the existing synchronization problems. Third, as a result of using the framework, our solutions will be composed of an architecture-independent part that relies only on the intrinsic requirements of the problem and an architecture-dependent part that achieves these requirements on a given architecture. This separation of the solutions into an architecture-independent part and an architecture-dependent part makes them modular and transportable across different architectures.

Given that a unifying framework is both needed and useful, how should one go about designing it? Our starting point was the recognition that every solution to a synchronization problem necessarily involves some form of arbitration among the processes wishing to synchronize, and that the basis for this arbitration is usually some values that are assigned to the processes upon request. We call these values *ranks*. Any solution to a synchronization problem is thus separated into two phases: computation of ranks, and arbitration based on the computed ranks. Depending upon the particular synchronization problem and its solution, these two phases may be carried out in a number of different ways. For example, consider a time-stamp-based solution to the mutual exclusion problem [24]. In such a solution every process that wishes to enter its critical section first obtains a time-stamp, thus executing the first phase. Next, the processes compare their time-stamps and the process with the lowest time-stamp enters its critical section. This is the second phase. As another example, consider a solution to the dining philosophers problem [10] in which a *hungry* philosopher sends out requests for forks and establishes its place in an acyclic graph, thus executing the first phase. Next, the *hungry* philosophers compare their places in the graph and the philosopher at the *top* of the graph *eats*. This is the second phase.

In order to separate the two phases of rank computation and arbitration based on the computed ranks, we propose a program module that carries out the first phase of rank computation. This module is implemented in a distributed manner as a collection of submodules; each process is assigned a local submodule that is responsible for computing the current rank of the process when one is needed. The submodules communicate with each other to ensure that the computed ranks for the processes meet some global requirements (to be discussed later). Henceforth, we refer to the submodule assigned to a process as its *local ranker*, and refer to the collection of these local rankers as the *ranker*.

When a process wishes to synchronize, it requests a rank from its local ranker and waits until this rank has been computed. After that the process compares its rank with the ranks of other competing processes and proceeds accordingly. When the process no longer needs its rank, it returns the rank to its local ranker. It is worth noting that the ordering of the ranks of the processes is

not pre-determined, instead it is defined by the order in which the processes request ranks. Later, we define a property, called *precedence*, that ensures this dynamic assignment of ranks.

The rest of this paper is organized as follows. In Section 2, we discuss the background for this work. Next, we present a brief overview of UNITY [10], the formalism that we use for presentation of the algorithms and the proofs. In Sections 4 and 5 we discuss the specification and classification of rankers. Then, we illustrate how synchronization problems are solved using rankers. We consider the FCFS doorway problem in Section 6, the dining philosophers problem in Section 7, and the resource allocation problem in Section 8. In Section 9 we present some implementations for the rankers classified in Section 5. Concluding remarks are in Section 10.

2. Background

The idea of rankers has its roots in the bakery algorithm that Lamport proposed as a solution to the mutual exclusion problem [22]. In this algorithm, a process that wishes to enter its critical section first obtains a ticket, then compares its ticket with those of other processes and enters its critical section when its ticket has the highest priority. Though this algorithm hints at a general abstraction of ranks and explicitly separates the two phases of rank computation and arbitration, it cannot be used as a general framework for solving all synchronization problems. First, the author's definition of ranks in the algorithm is specific to the problem being solved, namely mutual exclusion. As shown later, some synchronization problems require weaker definitions of ranks while yet others require stronger definitions of ranks. Second, the interface between the implementation of rankers (which is dependent on the underlying architecture) and their usage (which is independent of the underlying architecture) is not defined. A later paper by Lamport [23] does, however, explore the idea of an architecture-independent synchronization primitive that generalizes the conditional critical region [5]. The paper also discusses implementations of the primitive for the shared variable architecture.

The bounded time-stamps algorithm of Israeli and Li [21] and the colored tickets algorithm of Fischer et al. [18] also use a notion of ranks. Israeli and Li define a sequential time-stamp system and a concurrent time-stamp system and use the latter to construct a multi-writer atomic register. Fischer et al. solve the resource allocation problem (also called the K-mutual exclusion problem) using colored tickets, a generalization of tickets used in the bakery algorithm. All these characterizations of ranks are somewhat problem- and architecture-specific and, therefore, they cannot serve as a general framework for the study of the entire class of synchronization problems. More recently, Dolev and Shavit [15] and Dwork and Waarts [16] have presented another

implementation for bounded time-stamps that can be used to construct a multi-writer atomic register and to solve the K-mutual exclusion problem.

The idea of eventcounts and tickets proposed by Kanodia and Reed [35] was an earlier attempt at classifying synchronization problems. The authors solve the producer–consumer problem and the readers–writers problem using this abstraction. However, they do not state the properties enjoyed by eventcounts and sequencers; as a result the interface between the application and the implementation of these abstractions is not clear, and eventcounts and sequencers appear to be implementations rather than specifications. Moreover, the classification of synchronization problems that they obtain (one class corresponding to each of the two abstractions) is coarser than our classification.

3. The formalism

In this section we present a brief introduction to UNITY; this presentation is somewhat simplified as we discuss only those aspects that we use.

A program consists of a predicate describing the initial values of the variables and a set of assignment statements. An assignment statement consists of one or more assignment components separated by ‖. An assignment component is either an enumerated assignment or a quantified assignment. An enumerated assignment has a variable list on the left, a corresponding expression list in the middle, and a boolean expression on the right called the *guard* (which by default is *true*):

⟨variable-list⟩ := ⟨expression-list⟩ *if* ⟨guard⟩.

A quantified assignment specifies a quantification and an assignment that is to be instantiated with the given quantification; a quantification names a set of bound variables and a boolean expression (the range) satisfied by the instances of the bound variables. The set of assignment statements in a program is written down either by enumerating every statement singly and using [] as the set constructor, or by using a quantification of the form ⟨[] *var* : *range* :: *statement*⟩. Symbol [] is called the *union* operator.

A program execution starts from any state satisfying the initial conditions and goes on forever; in each step of execution some assignment statement is selected nondeterministically and executed. Nondeterministic selection is constrained by the following fairness rule: Every statement is selected infinitely often [10]. An assignment statement is executed by executing all of its assignment components simultaneously. An assignment component is executed by first evaluating all expressions and the boolean guard and then assigning the values of the evaluated expressions to the appropriate variables if the associated guard is *true*; otherwise, the variables are left unchanged.

Program properties are expressed using four relations on predicates—**unless**, **invariant**, **ensures**, and **leads-to**. The first two are used for stating safety properties whereas the last two are used for stating progress properties.

For any two predicates p and q, the property p **unless** q holds in a program iff for all statements s in the program the following Hoare triple [20] holds

$$\{p \wedge \neg q\}\ s\ \{p \vee q\}.$$

Informally, if p is true at some point in the computation, then either q never holds and p holds forever from this point on, or q holds eventually and p continues to hold until q holds. As an example consider $x = k$ **unless** $x > k$, for all k, which states that x never decreases. As another example consider *thinking.u* **unless** *hungry.u* which states that philosopher u remains thinking until becoming hungry.

For any predicate p, the property **invariant** p holds in a program iff p holds initially and the program never falsifies p, i.e.,

initially p $\wedge$ p **unless** *false*.

As an example consider **invariant** $x \geqslant 5$ which states that variable x is always greater than 4.

For any two predicates p and q, the property p **ensures** q holds in a program iff p **unless** q holds in the program and there exists a statement s in the program such that

$$\{p \wedge \neg q\}\ s\ \{q\}.$$

Thus, if p is true at some point in the computation then q holds eventually and p continues to hold until q holds.

The relation **leads-to** is denoted $\mapsto$, and is defined to be the strongest relation satisfying the following three rules.

- $(p$ **ensures** $q) \Rightarrow (p \mapsto q)$,
- $((p \mapsto q) \wedge (q \mapsto r)) \Rightarrow (p \mapsto r)$, and
- for any set W, $(\forall m : m \in W : p.m \mapsto q) \Rightarrow ((\exists m : m \in W : p.m) \mapsto q)$.

The first two rules imply that $\mapsto$ includes the transitive closure of **ensures** and the third rule allows us to induct over sets. Given that $p \mapsto q$ in a program, we can assert that once p becomes *true*, eventually q becomes *true*. However, unlike p **ensures** q, we cannot assert that p will remain *true* as long as q does not become *true*. As an example of this property consider *hungry.u* $\mapsto$ *eating.u* which states that a hungry philosopher u eventually eats. As another example consider *send.m* $\mapsto$ *receive.m* which states that if a message m is sent, it is eventually received.

Programs are composed by taking the union of their assignment statements. The union of two programs F and G is denoted $F \,[]\, G$.

4. Specification of rankers

In this section we specify the interface between a user process and its local ranker; we also state the properties that need to be satisfied by the ranker, i.e., by the collection of all local rankers. The interface between a user process and its local ranker is specified in Section 4.1, and properties of the ranker are specified in Section 4.2.

4.1. The interface

The interface between a user process u and its local ranker consists of two shared variables—*state.u* and *r.u*. Variable *state.u* denotes the state of process u with respect to its ranking and can take any one of three possible values—*white*, *grey*, or *black*. If *state.u* is *white*, then process u is not interested in synchronization and so does not need a rank; if *state.u* is *grey*, then process u is interested in synchronization and wants a new rank; if *state.u* is *black*, then process u has obtained a new rank. Initially, this variable is *white* and it assumes the three values in the order, *white*, *grey*, and *black*. The transitions from *white* to *grey* and from *black* to *white* occur in the user processes whereas the transition from *grey* to *black* occurs in ranker. (For readers familiar with Lamport's bakery algorithm [22], we note the following correspondences: *white* ≡ *idle*, *grey* ≡ *choosing*, and *black* ≡ *active*.)

Variable *r.u*, the other variable in the interface between process u and its local ranker, denotes the rank of process u. This variable can be modified only by the local ranker; moreover, all such modifications are made only when the state of process u is *grey* (i.e., it is constant when the state of process u is *white* or *black*). A binary irreflexive relation $\prec$ on the values of *r.u* is used for comparing the ranks of different processes: $r.u \prec r.v$ denotes that process u has a lower rank than process v. We require that the relation $\prec$ satisfy the following condition for every u and v:

$$\neg(r.u \prec r.v \ \wedge \ r.v \prec r.u),$$

i.e., both u and v cannot have a rank lower than the other. (Note that $\prec$ is not necessarily a partial order.) This completes the specification of the interface between a user process and its local ranker.

4.2. Specification of the ranker

In order to motivate the specification of the ranker, we first outline the protocol between the user processes and the ranker. When an user process wants to synchronize with other processes, it sets its state to *grey* and waits for a rank from the ranker. The ranker computes a current rank for the process, then sets the state of the process to *black*. After this, the process compares

its rank with the ranks of other processes using the relation $\prec$ and proceeds accordingly. This comparison is problem-specific and the responsibility of the processes. When the process no longer needs the rank (i.e., has performed the needed synchronization), the process sets its state to *white*. And the cycle continues. (Note that there are no constraints on the number of processes that may be in a given state at a time; in particular, it is possible for more than one process to be *grey* at a time.) All the algorithms that we consider in this paper are structured as specified above.

By examining the synchronization problems and their solutions, we have identified five useful properties of rankers—*responsiveness, precedence, acyclicity, comparability,* and *stability*—which are discussed next. In choosing these properties, we have strived for simplicity and completeness. Each one of these properties addresses an orthogonal aspect of synchronization and is motivated by a particular class of synchronization problems.

Notation. We group all the user processes together in a program called *user*. All program properties are of the composite program *user* [] *ranker*. For convenience, we define the following three predicates for every u.

$$\begin{aligned} white.u &\equiv state.u = white, \\ grey.u &\equiv state.u = grey, \\ black.u &\equiv state.u = black. \end{aligned}$$

Responsiveness

A basic property that all rankers must have is that they should compute ranks for requesting processes (otherwise, a process that wishes to synchronize may starve forever). This requirement is captured by responsiveness which states that if a process requests a rank, then its local ranker eventually responds by computing a rank, i.e., every *grey* process eventually becomes *black*. Formally, for every process u,

$$grey.u \mapsto black.u.$$

The above requirement that a *grey* process eventually becomes *black* is inadequate by itself. We also impose the additional requirement that the transition from *grey* to *black* be achieved by executing a finite number of wait-free statements. Though essential, the additional requirement is difficult to state formally in UNITY. So, instead of cluttering the exposition here, we address this issue further in Section 10.

Precedence

Another basic property that all rankers must have is that computed ranks should be dynamic and dependent on the order in which the processes request ranks (otherwise, trivial rankers in which the rank of a process never changes

can be defined). The simplest possible way to relate the ranks of the processes to the order in which they request ranks is a first-come, first-served scheme. Unfortunately, a ranker that assigns the ranks based on the order in which the processes turn *grey* (i.e., request for ranks) or based on the order in which the processes turn *black* (i.e., receive ranks) cannot be implemented. Therefore, we choose the *time interval* over which a process is *grey* as the basis for assigning ranks. Specifically, if the last granting of a rank to process u precedes the last request of a rank by process v, then process u has a higher rank[1] than process v. The property of precedence is stated as follows: for every distinct u and v

$$\textbf{invariant}\ \mathit{black.u} \wedge \mathit{black.v} \wedge \mathit{precedes.u.v} \Rightarrow r.v \prec r.u,$$

where *precedes.u.v* is an auxiliary boolean variable that captures the order in which processes u and v change their states; it is defined shortly. This property is illustrated in Fig. 1.

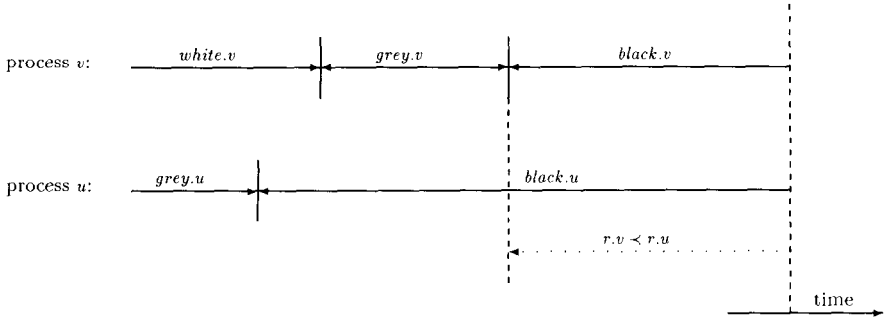

Fig. 1. Property of precedence.

The predicate (*black.u* ∧ *black.v* ∧ *precedes.u.v*) is *true* iff both processes u and v are *black* and process u transits from *grey* to *black* before process v transits from *white* to *grey* (i.e., the last granting of a rank to process u precedes the last request for a rank by process v). The condition of precedence states that, in that case, process u has a higher rank than process v. (Observe that the property of precedence does not relate the rank of a *black* process with the rank of a *grey* process even if the predicate *precedes.u.v* is true. We will discuss a property called stability later that will impose some requirements in this case.)

Like all auxiliary variables, variable *precedes.u.v* is not implemented; it is used only in the specification and proof of correctness. Its formal definition

[1] Unlike time-stamps [24], we associate *older* with *higher* in this paper.

which appears next may be skipped without loss of continuity. Based on an idea due to Misra [29], *precedes.u.v* is defined by the following two assertions.

initially $\neg precedes.u.v$

and

$$state.u = x \wedge state.v = y \wedge precedes.u.v = b \textbf{ unless}$$
$$\neg(state.u = x \wedge state.v = y) \wedge$$
$$(precedes.u.v \equiv black.u \wedge (white.v \vee b)).$$

The first assertion states that *precedes.u.v* is *false* initially and the second assertion defines how *precedes.u.v* changes with each change in the states of processes u and v. It is set to *true* if process v becomes *white* while process u is *black*. Once *true*, it continues to remain *true* until process u becomes non-*black* and then it is set to *false*. It is possible to show based on the above definition that the relation *precedes* is irreflexive and transitive.

Acyclicity

The two properties of rankers that we have stated so far are either local to one process or pairwise over processes. For a number of synchronization problems, this is insufficient as some global properties on the ranks of all processes are required. We choose acyclicity as such as a global property and state it formally by requiring that the transitive closure of $\prec$, denoted by $\prec^*$, be irreflexive, i.e., for all u,

invariant $\neg(r.u \prec^* r.u)$.

This property is not mandatory of all rankers as some synchronization problems (discussed in the next section) can be solved without it.

Comparability

For some synchronization problems like the mutual exclusion problem, a total order on the ranks seems essential. In order to achieve such a total order, we introduce the property of comparability, which along with acyclicity gives us an irreflexive total order on the ranks. Comparability states that the ranks of any two processes can be compared, i.e., for every distinct u and v,

invariant $r.u \prec r.v \ \vee \ r.v \prec r.u$.

The proof that this property together with acyclicity implies an irreflexive total order is as follows. Let u, v, and w be any three distinct processes such that $r.u \prec r.v$ and $r.v \prec r.w$. On account of comparability, $r.u \prec r.w$ or $r.w \prec r.u$. But, because $\prec$ is acyclic, $\neg(r.w \prec r.u)$; therefore, $r.u \prec r.w$. This means that $\prec$ is transitive, and therefore an irreflexive total order.

Stability

Consider two processes u and v such that process u obtains a rank before process v requests a rank. Consider a state in which process u is *black* and wishes to compare its rank with that of process v. Since the rank of a *grey* process may change, this comparison of ranks is meaningful only when the state of process v is *white* or *black*. If the state of process v is *black*, then, due to precedence, the rank of v will be less than the rank of u. If the state of process v is *white*, then process u can assume that it has a higher rank because when process v does obtain a rank, its rank will be lower than the rank of u. But, if the state of process v is *grey*, then no meaningful comparison of ranks is possible and, therefore, process u must wait until the state of process v settles to *white* or *black*. For some synchronization problems like mutual exclusion, this waiting period turns out to be bounded and process u will eventually find the state of process v to be non-*grey* and be able to compare the ranks meaningfully. However, for synchronization problems in which process v can time out from *black* to *white* right away, process u may always find the state of v to be *grey* and thus, unable to compare ranks, may starve forever. We introduce the property of stability in order to solve this problem.

The property of stability states that if a *black* process u has a higher rank than another process (no matter what the state of the other process is, *white*, *grey*, or *black*) then it continues to have a higher rank until u becomes *white*. Formally, for every distinct u and v,

$$black.u \wedge r.v \prec r.u \textbf{ unless } white.u.$$

Repeated timeouts by process v no longer pose a problem because now process u can compare its rank with the rank of process v meaningfully regardless of the state of v. In fact, if process v repeatedly cycles through its states while process u is waiting, then eventually u will have a higher rank than v (due to precedence), and then it will continue to have a higher rank (due to stability).

5. Classification of rankers

In the previous section, we discussed five properties of rankers—responsiveness, precedence, acyclicity, comparability, and stability. By combining these five properties in all possible ways, we obtain four useful rankers in a strict hierarchy. The first ranker in this hierarchy, ranker Φ, satisfies the responsiveness and precedence properties; the other three rankers in the hierarchy are obtained by adding the remaining three properties—acyclicity, comparability, and stability successively. Thus, we get ranker A (for acyclicity), ranker C (for comparability), and ranker S (for stability). This hierarchy of rankers and the class of synchronization problems that they solve is shown in Fig. 2.

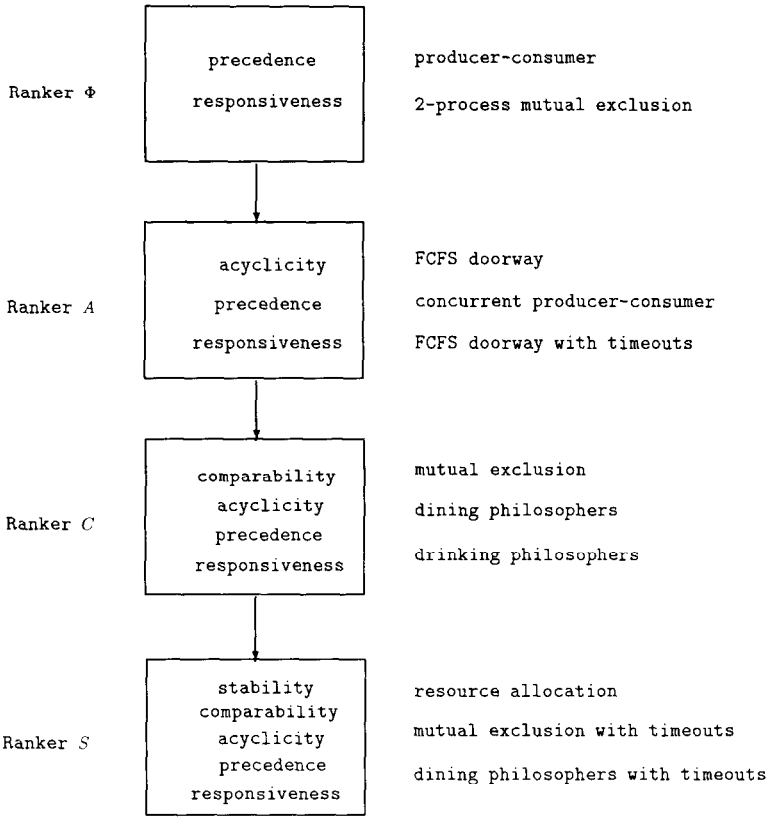

Fig. 2. Classification of rankers and synchronization problems

As we proceed from the top of this hierarchy to the bottom, the rankers become stronger, i.e., they satisfy more properties. However, depending on the underlying architecture, their implementations also may become more difficult (especially if one considers implementations using bounded variables). Therefore, though ranker S can be used to solve any problem the others can solve, the problem at hand may not require all the properties of ranker S. So, we may be better off using another ranker that is weaker than ranker S, and may be easier to implement. Given a particular synchronization problem, it is possible to associate it with the weakest ranker that can be used to solve it. Such a classification of synchronization problems and the ranker associated with each such class is discussed next. Parts of this classification have also been observed by other researchers [35].

5.1. *Ranker Φ*

This ranker solves synchronization problems that do not require properties involving the ranks of more than two processes. This is because the properties

satisfied by this ranker, responsiveness and precedence, are, respectively, local and pairwise properties on process ranks. Examples of problems that are solved by this ranker are the producer–consumer problem [5] and the two-process mutual exclusion problem [31]. Observe that if the requests for ranking by the processes are serial, i.e., *precedes* is a total order, then precedence alone guarantees a total order on the ranks of the *black* processes. As a result, this ranker corresponds to the "sequential time-stamp system" discussed by Israeli and Li [21]. Therefore, the implementations that they propose for sequential time-stamp systems can also be used for implementing this ranker.

5.2. Ranker A

Acyclicity, a global property over all processes, guarantees the absence of cycles on the ranks of the processes. However, it allows any number of processes to have maximal ranks. Therefore, this ranker solves problems that require global properties over ranks of processes but do not limit the number of processes that are performing the critical action at any time. Examples of such problems are the first-come, first-served doorway problem [18,22,26] and the concurrent producer–consumer problem [37]. This ranker corresponds to the mechanism of "eventcounts" proposed by Kanodia and Reed [35]. Though eventcounts satisfy the stronger condition of a partial order, the class of problems that eventcounts can solve appears to be the same as the class of problems that can be solved by this ranker.

5.3. Ranker C

As seen earlier, the property of comparability along with acyclicity makes $\prec$ an irreflexive total order on all ranks. Therefore, this ranker solves problems that require a total order on the processes. Examples of such problems are the mutual exclusion problem [13] (requires a total order on all processes), the dining philosophers problem [14] (requires a total order on neighboring processes), and the drinking philosophers problem [9] (requires a total order on neighboring processes with conflicting requests). Because this ranker guarantees a total order on the ranks of the processes, it corresponds to the abstraction of time-stamps by Lamport [24], Dolev and Shavit [15], and Dwork and Waarts [16], and the mechanism of "sequencers" proposed by Kanodia and Reed [35]. It is not quite clear whether the abstraction of "bounded time-stamps" proposed by Israeli and Li [21] corresponds to ranker *A* or ranker *C*. This is because though their specification of "bounded time-stamps" requires a total order on the time-stamps, their implementation does not seem to enforce this requirement.

5.4. Ranker S

None of the previous three rankers allow for a meaningful comparison of ranks between processes when one of them is *grey*. This is because the rank of a *grey* process may change. For some synchronization problems like the mutual exclusion problem, this is not a hindrance as a *grey* process eventually turns *black* and the ranks of *black* processes can be compared. However, in some other synchronization problems like the resource allocation problem, it is possible for processes to be continuously cycling through their states, and thus some process may always find the state of another process to be *grey* and wait forever. (This is also true of synchronization problems where a process may time out or fail in the state that it is waiting to synchronize, and make a direct transition to a state in which it is not waiting to synchronize, thus bypassing the state in which it synchronizes or performs the critical action). In order to solve these problems we use ranker S. This is because the property of stability in conjunction with the property of precedence ensures that if a process v continuously cycles through its states while another process u is waiting, then eventually u has a higher rank than v (due to precedence), and then it continues to have a higher rank (due to stability). Examples of problems that are solved by this ranker are the resource allocation problem, the mutual exclusion problem with timeouts, the dining philosophers problem with timeouts, and the drinking philosophers problem with timeouts. This ranker appears to be new and does not correspond to any of the proposed abstractions in the literature.

In the next three sections, we discuss how to solve three specific synchronization problems using rankers.

6. FCFS doorway

The FCFS (first-come, first-served) doorway, first proposed by Lamport as a part of the bakery algorithm [22], has been used in a number of different mutual exclusion algorithms [26,33]. Its usefulness stems from the fact that it can be combined with a mutual exclusion solution that guarantees global progress, to yield a composite solution that guarantees individual progress. In other words, suppose we have a mutual exclusion algorithm that guarantees mutually exclusive access (i.e., no two processes are in their critical section at the same time) and global progress (i.e., if there are some processes waiting to enter their critical section, then some process eventually enters its critical section). Then, we can compose this algorithm with an FCFS doorway so that the composite program ensures individual progress (i.e., every waiting process eventually enters its critical section) in addition to mutually exclusive access. We present the specification of the problem in Section 6.1, discuss our solution using rankers in Section 6.2, and sketch a proof of its correctness in Section 6.3.

6.1. Problem specification

We have a set of processes $\{u, v, \ldots\}$. Associated with every process u is a variable *mode.u* that can take any one of four possible values—*out*, *at*, *wait*, or *enter*. Initially, *mode.u* = *out*, for each u. For convenience, we define the following four predicates for each u.

$$\begin{aligned} out.u &\equiv mode.u = out, \\ at.u &\equiv mode.u = at, \\ wait.u &\equiv mode.u = wait, \\ enter.u &\equiv mode.u = enter. \end{aligned}$$

Variables *mode.u* cycle through the values, *out*, *at*, *wait*, and *enter* in that order; the transitions from *out* to *at* and from *enter* to *out* are determined by the process while the transitions from *at* to *wait* and *wait* to *enter* are determined by the solution. The intuition behind the four states is as follows. If the state of a process is *out*, then the process is not interested in entering the doorway; if the state of a process is *at*, then the process is interested in entering the doorway (the transition from *out* to *at* is made internally by the process); if the state of a process is *wait*, then the process is waiting to enter the doorway (the transition from *at* to *wait* is made by the solution and corresponds roughly to the few lines of code that a process typically executes in order to "enter" the doorway in the FCFS mutual exclusion algorithms); if the state of a process is *enter*, then it has entered the doorway (it is in this state that a process typically participates in mutual exclusion).

We elaborate on the above specification by comparing it with the following process skeleton in [26].

```
repeat forever
    out
    noncritical section;
      at
    FC trying;
      wait
    FC critical section;
      enter
      ME trying;
      ME critical section;
      ME exit;
    FC exit;
end repeat
```

Here *FC* represents a solution to the FCFS doorway problem, *ME* represents a solution to the mutual exclusion problem, and the above process skeleton rep-

resents the composition of the two solutions which obtains individual progress from global progress. The state of a process corresponding to a particular point in its execution is shown in braces. An *out* to *at* transition is made when a process exits its *noncritical section* and enters the *FC trying* section. An *at* to *wait* transition is made when a process exits its *FC trying* section. A *wait* to *enter* transition is made when a process enters its *FC critical section*. Finally, an *enter* to *out* transition is made when a process exits its *FC exit* section. The goal is to write program code for the transitions *at* to *wait* and *wait* to *enter* such that the composite program satisfies the following three properties. For all distinct u and v,

$$\begin{array}{ll} \{no\ waiting\} & at.u \mapsto wait.u, \\ \{no\ deadlock\} & (\exists u :: wait.u) \mapsto (\exists u :: enter.u), \\ \{FCFS\} & \textbf{invariant}\ \neg(before.u.v \wedge enter.v), \end{array}$$

where *before.u.v* is an auxiliary boolean variable that captures the order in which processes u and v change their states. It is *true* iff process u transits from *at* to *wait* before process v transits from *out* to *at*.

The no waiting condition states that a process that is *at* the doorway, eventually transits to the *wait* state. This transition corresponds to the few lines of code that a process typically executes in existing solutions to the problem. (We should actually require that the transition from *at* to *wait* happens in a wait-free manner. However, this is difficult to state formally and so we adhere to the weaker eventuality requirement.) The no deadlock condition states that if there exists some *waiting* processes then eventually some process will *enter* the doorway. The FCFS condition states that if the predicate *before.u.v* holds, i.e., process u completed its *at* to *wait* transition before process v completed its *out* to *at* transition, then process v does not *enter* the doorway until process u goes *out* of the doorway.

Auxiliary variable *before.u.v* is defined by the following two assertions.

$$\textbf{initially}\ \neg before.u.v$$

and

$$\begin{array}{l} mode.u = x \wedge mode.v = y \wedge before.u.v = b\ \textbf{unless} \\ \neg(mode.u = x \wedge mode.v = y) \wedge \\ (before.u.v \equiv (wait.u \vee enter.u) \wedge (out.v \vee b)). \end{array}$$

The first assertion states that *before.u.v* is *false* initially and the second assertion defines how *before.u.v* changes with each change in the states of processes u and v. It is set to *true* if the predicate *wait.u* $\vee$ *enter.u* holds simultaneously with the predicate *out.v* (i.e., the state of process u is *wait* or *enter* while the

state of process v is *out*). Once *true*, it continues to remain *true* until process u goes *out* of the doorway, and then it is set to *false*.

6.2. *Solution*

In order to satisfy the FCFS condition, we design our solution so that if the predicate *before.u.v* holds then so does the predicate *precedes.u.v*, i.e., if process u transits from *at* to *wait* before process v transits from *out* to *at* (thus, setting *before.u.v* to *true*) then process u also transits from *grey* to *black* before process v transits from *white* to *grey* (thus, setting *precedes.u.v* to *true*). Our solution is designed so that process v does not *enter* the doorway if its rank is less than the rank of any other *black* process. Then, on account of precedence, and because $before.u.v \Rightarrow precedes.u.v$, if processes u and v are *black* and *before.u.v* holds, then the rank of v will be less than the rank of u, and consequently, process v will not *enter* the doorway until process u goes *out* of the doorway. This ensures that our solution meets the FCFS condition.

In order to satisfy the no deadlock condition, there should always be at least one *black* process with a maximal rank. This implies that the ranks of the *black* processes should be acyclic. Therefore, we solve this problem using ranker A (ranker Φ does not suffice as it places no global constraints on the ranks of the processes). The program for process u is as follows. Variable *checked.u.v* is a local variable and predicate *high.u.v* is defined to be $\neg(black.v \wedge r.u \prec r.v)$. The final statement in the program denotes an assignment statement in which *mode.u* is assigned *out*, *state.u* is assigned *white*, and each *checked.u.v* is assigned *false*, provided the guard *enter.u* holds. Since all the *checked.u.v* variables are local to process u, this multiple assignment statement does not assume high atomicity.

initially $white.u \wedge out.u \wedge (\forall v :: \neg checked.u.v)$
assign

$mode.u, state.u := at, grey$	**if** $out.u$
$[]$ $mode.u := wait$	**if** $at.u \wedge black.u$
$[]$ $\langle [] v :: checked.u.v := true$	**if** $wait.u \wedge high.u.v \rangle$
$[]$ $mode.u := enter$	**if** $wait.u \wedge (\forall v : v \neq u : checked.u.v)$
$[]$ $mode.u, state.u := out, white$	**if** $enter.u$
$\|\|$ $\langle \|\| v :: checked.u.v := false$	**if** $enter.u \rangle$

end

Process u manipulates variables *state.u* and *mode.u* as follows. When it sets *mode.u* to *at*, it also sets *state.u* to *grey* (to request a rank). Then, it waits for *state.u* to become *black* (this happens eventually due to responsiveness) and sets *mode.u* to *wait*. Then, it compares its rank asynchronously with other

processes and sets boolean variable *checked.u.v* to *true* if process v is either non-*black* or does not have a higher rank than u (this is in order to ensure the FCFS condition). When all the variables *checked.u.v* are *true*, the process transits from *wait* to *enter* state. Finally, when it sets *mode.u* to *out*, it also sets *state.u* to *white* and resets each variable *checked.u.v* to *false*.

6.3. *Proof of correctness*

A formal proof of correctness for this solution is detailed in [37]; we present a sketch of this proof. Recall that ranker A satisfies the properties of responsiveness, precedence, and acyclicity. These three properties are used to prove that our solution satisfies the no waiting, no deadlock, and FCFS properties. The proof of no waiting is based on responsiveness, the proof of no deadlock is based on responsiveness and acyclicity, and the proof of FCFS is based on precedence.

Our solution satisfies the no waiting condition because of the following reason. A process which is *at* the doorway sets its state to *grey* and waits for this state to become *black*. This happens eventually because of responsiveness of ranker A. Once the state becomes *black*, the process makes the required transition to the *wait* state.

To see why our solution satisfies the no deadlock condition consider the processes which are in the *wait* state at any time. Because of the responsiveness and acyclicity properties of ranker A, eventually one of these processes has a rank no smaller than all other *black* processes, i.e., for some process u, the predicate *high.u.v* holds for all other v. Therefore, either this process makes a transition to the *enter* state or some other process starts *waiting* with a higher rank replacing u as the process with a rank no smaller than all other *black* processes. But, events of the latter kind are bounded by the number of processes. Therefore, some *waiting* process will eventually go to the *enter* state, thus satisfying the no deadlock condition.

Our solution satisfies the FCFS condition because of the following reason. If *before.u.v* holds, i.e., a process u transits from *at* to *wait* before another process v transits from *out* to *at*, then process u also requests and obtains a rank before v. As a result, because of precedence, process v has a lower rank than process u, i.e., $r.v \prec r.u$. Therefore, when process v starts *waiting*, the predicate *high.v.u* will be *false* (as process u is *black* and $r.v \prec r.u$) and consequently, it does not *enter* the doorway until the predicate *high.u.v* becomes *true*, i.e., until process u goes *out* of the doorway.

7. Dining philosophers

The dining philosophers problem is a paradigm for conflict resolution [14]. The problem consists of a number of processes placed on the vertices of a

graph that may wish to enter their critical section from time to time (if this graph is fully connected, then this problem reduces to the mutual exclusion problem). We have to design a solution that ensures mutual exclusion (i.e., that no two neighboring processes are in their critical sections at the same time) and freedom from starvation (every process wishing to enter its critical section eventually gets to enter it). We present the specification of the problem in Section 7.1, discuss our solution in Section 7.2, and sketch a proof of its correctness in Section 7.3. Finally, in Section 7.4 we consider a variation of the problem in which a hungry process may time out and transit directly to the thinking state.

7.1. Problem specification

We have a set of processes $\{u, v, \ldots\}$ and a symmetric, irreflexive relation N defined on this set. Processes u and v are said to be *neighbors* iff $N.u.v$ is true. Associated with every process u is a variable $mode.u$ that can take any one of three possible values—t (thinking), h (hungry), or e (eating). Initially, $mode.u = t$, for each u. For convenience, we define the following three predicates for every u:

$$\begin{aligned} t.u &\equiv mode.u = t, \\ h.u &\equiv mode.u = h, \\ e.u &\equiv mode.u = e. \end{aligned}$$

Variables $mode.u$ cycle through the values, t, h, and e in that order; the transitions from t to h and from e to t are determined by the process while the transition from h to e is determined by the solution. It is given that all eating periods are finite, i.e., $(\forall u :: e.u \mapsto t.u)$.

It is required that the composite program satisfy the following two properties. For all distinct u and v,

$$\begin{aligned} &\{mutual\ exclusion\} \quad \textbf{invariant}\ \neg(e.u \wedge e.v \wedge N.u.v), \\ &\{no\ starvation\} \quad\quad h.u \mapsto e.u. \end{aligned}$$

The mutual exclusion condition states that no two neighboring processes may eat at the same time. The no starvation condition states that every hungry process eventually eats.

7.2. Solution

We use ranks to arbitrate between competing philosophers: a hungry process u eats only if it has a higher rank than all its competing neighbors. Thus, in order to satisfy the mutual exclusion condition, the ranks of any set of neighboring processes should be acyclic (otherwise, two neighbors may be in their critical section at the same time). Also, in order to satisfy the no

starvation condition, the ranks of any two neighboring processes should be comparable (otherwise, neither process may have a rank higher than the other and the two processes end up in a deadly embrace waiting for each other forever). Due to the above two requirements of acyclicity and comparability, we solve this problem using ranker C.

The program for process u is as follows. Predicate *high.u.v* is defined to be $white.v \vee (black.v \wedge r.v \prec r.u)$. In other words, predicate *high.u.v* is *true* iff process v is not interested in entering its critical section (i.e., process v is *white*) or if process v is *black* and has a lower rank than process u. Observe that if the state of process v is *grey* then this predicate is *false*. In the following description, process v is assumed to range over the neighbors of process u.

initially $white.u \wedge t.u \wedge (\forall v :: \neg checked.u.v)$
assign
$mode.u, state.u := h, grey$ **if** $t.u$
$[] \langle [] v :: checked.u.v := true$ **if** $h.u. \wedge black.u \wedge high.u.v \rangle$
$[] mode.u := e$ **if** $(\forall v :: checked.u.v)$
$[] mode.u, state.u := t, white$ **if** $e.u$
$\| \langle \| v :: checked.u.v := false$ **if** $e.u \rangle$
end

Process u manipulates variables *state.u* and *mode.u* as follows. When it sets *mode.u* to h, it also sets *state.u* to *grey* (to request a new rank). Then, it waits for *state.u* to become *black* (this happens eventually due to responsiveness). After that, it compares its rank asynchronously with all neighboring processes and sets boolean variable *checked.u.v* to *true* if process v is either *white* (in which case, on account of precedence, process v will have a lower rank than process u when it becomes hungry) or is *black* and has a lower rank. If process u finds a neighboring process to be *grey*, then, because the rank of a *grey* process may change, it waits until that process becomes non-*grey*. (However, due to responsiveness, a *grey* process eventually becomes *black* and hence, process u is eventually able to proceed.) When all the variables *checked.u.v* are *true*, the process transits from the hungry to the eating state. Finally, when process u completes eating and sets *mode.u* to t, it also sets *state.u* to *white*, and resets each variable *checked.u.v* to *false*.

7.3. *Proof of correctness*

A formal proof of correctness for this solution is detailed in [37]; we present a sketch of this proof. Recall that ranker C satisfies the properties of responsiveness, precedence, acyclicity, and comparability. These four properties are used to prove that our solution satisfies mutual exclusion and no starvation.

The proof of mutual exclusion is based on precedence while the proof of no starvation is based on all four properties—responsiveness, precedence, acyclicity, and comparability.

Our solution satisfies the mutual exclusion condition because it satisfies the following invariant (due to precedence of ranker C)—an eating process is *black*, and has a higher rank than all its *black* neighbors. It follows from this invariant that for two neighboring processes to be eating at the same time, each has a rank higher than the other, a condition that is prohibited on account of the definition of ranks. Therefore, two neighboring process do not eat at the same time.

To see why our solution is free from starvation, consider any process u that is hungry and let v be any neighboring process. Because of the responsiveness and comparability properties of ranker C, eventually either the predicate *high.u.v* holds (in which case process u does not wait for process v) or process v becomes *black* and has a higher rank than u (in which case process u waits for process v). In the latter case, consider the directed *wait-for* graph rooted at u. Because of acyclicity of ranker C, this graph is acyclic, and therefore, has some leaf nodes. Eventually, each process at a leaf node will start eating and will later start thinking (because all eating periods are finite). This reduces the size of the *wait-for* graph (due to the precedence and responsiveness properties). It follows by induction on the size of this graph that it eventually contains no edges, i.e., process u is not waiting for any other process. When this happens, process u will make a hungry to eating transition, thus ensuring no starvation.

7.4. Dining philosophers with timeouts

In the dining philosophers problem we discussed earlier, a hungry process cannot make a direct transition to thinking without eating first (as $h.u$ **unless** $e.u$). In some situations, this may be undesirable and we may want to allow a hungry process that has waited long enough to time out, and make a direct transition to the thinking state. Such a direct transition is also desirable if we are modeling process failures and assume that a failed process returns to its initial state, the thinking state. These two considerations motivate the problem that we discuss next.

As before, we have a set of processes $\{u, v, \ldots\}$, a symmetric and irreflexive relation N that defines neighborhood, and a variable *mode.u* associated with a process u. The set of values that this variable takes and the order in which these values are assumed are as before. Once more, all eating periods are assumed to be finite. It is required that the composite program satisfy the following two properties for all distinct u and v,

$$\{\textit{mutual exclusion}\} \quad \textbf{invariant } \neg(e.u \wedge e.v \wedge N.u.v),$$
$$\{\textit{no starvation}\} \quad h.u \mapsto e.u \vee t.u.$$

The mutual exclusion condition is same as before; however, the no starvation condition has been weakened in order to allow the possibility of a timeout.

Contrary to the previous version of the problem (where processes do not time out), this problem cannot be solved by ranker C. To see why, consider any two neighboring processes u and v. Assume that process u is hungry and *black* and is waiting for the predicate *high.u.v* (defined to be $white.v \vee (black.v \wedge r.v \prec r.u)$) to become *true* in order to enter its critical section. Also, assume that process v is continuously timing out, i.e., cycling through the states t and h. Then, it is possible that each time process u attempts to compute the predicate *high.u.v*, the state of process v is *grey*, and consequently, *high.u.v* is *false*. Therefore, process u may remain hungry forever and thus, the no starvation requirement is not met.

The problem arises in the above scenario because of the definition of the predicate *high.u.v*: it is *false* if process v is *grey*. So, we need to redefine this predicate so that if process v continuously times out, then the predicate is set to *true* and remains *true* until process u eats. Consequently, we use ranker S, and redefine the predicate *high.u.v* to be $white.v \vee r.v \prec r.u$ (in other words, the predicate is *true* iff process v is not interested in its critical section or if it has a lower rank than process u). Then, if process v continuously times out, then this predicate will be set to *true* (due to precedence) and remain *true* until process u eats (due to stability).

The solution is similar to the previous solution; the only difference being that a process that is hungry and *black* can now make a direct transition to the thinking state without eating first. (A hungry process that is *grey* is not allowed to time out; however, because of the responsiveness condition of the ranker every such *grey* process eventually becomes *black* and thus, is able to time out). The program for process u is as follows. Predicate *high.u.v* is defined to be $white.v \vee r.v \prec r.u$.

initially $white.u \wedge t.u \wedge (\forall v :: \neg checked.u.v)$
assign
$mode.u, state.u := h, grey$ **if** $t.u$
$[]\ \langle []\ v :: checked.u.v := true$ **if** $h.u. \wedge black.u \wedge high.u.v \rangle$
$[]\ mode.u := e$ **if** $(\forall v :: checked.u.v)$
$[]\ mode.u, state.u := t, white$ **if** $e.u \vee (h.u \wedge black.u)$
$\|\ \langle \|\ v :: checked.u.v := false$ **if** $e.u \vee (h.u \wedge black.u) \rangle$
end

The proof of correctness of this solution is similar to the proof of correctness of the previous solution. The proof of mutual exclusion is based on the precedence and stability properties while the proof of no starvation is based on all the five properties of ranker S—responsiveness, precedence, acyclicity, comparability, and stability. The proofs appear in [37].

8. Resource allocation

The resource allocation problem (or the K-mutual exclusion problem [18]) is a generalization of the mutual exclusion problem [13]. In this problem, there are K identical copies of some resource, and a set of processes that contend for a copy of the resource. We are required to design a solution that ensures mutual exclusion (no more than K processes are accessing the resource at any one time) and no starvation (if a process wants a resource, it eventually gets one).

This problem can be easily solved using a solution to the mutual exclusion problem by maintaining a global queue of all the processes that desire to obtain a copy of the resource. Once a process wishes to obtain a copy, it places itself at the rear of the queue; once it comes within K of the head of the queue, it obtains a copy and when it is finished, it removes itself from the queue. All manipulations of the queue are carried out in exclusion by using the solution to the mutual exclusion problem. However, the above solution suffers from three drawbacks. The first drawback is the use of a global data structure. The second drawback is the loss of concurrency as there may be a lot of unnecessary contention for the queue. The third drawback is that even if one process fails in its critical section, the whole system comes to a halt. In this section, we present a solution with none of these drawbacks.

We present the specification of the problem in Section 8.1, discuss our solution in Section 8.2, and sketch a proof of its correctness in Section 8.3.

8.1. Problem specification

The specification of this problem is similar to the specification of the dining philosophers problem. However, now the *neighbors* relation N is complete, i.e., any two processes are neighbors. The definition of variable *mode.u* and the values it takes—t, h, and e—is as before. Once more all eating periods are finite, i.e., $(\forall u :: e.u \mapsto t.u)$.

It is required that the composite program satisfy the following two properties. For every u,

$$\{\textit{mutual exclusion}\} \quad \textbf{invariant } (\#u :: e.u) \leqslant K,$$
$$\{\textit{no starvation}\} \quad h.u \mapsto e.u.$$

The mutual exclusion condition states that at any time at most K processes are accessing the shared resource.[2] The no starvation condition states that every process that is waiting to access a resource eventually obtains a copy of the resource. Observe that when $K = 1$, we have the usual mutual exclusion problem.

[2]The expression $(\#u :: e.u)$ denotes the number of processes for which the predicate $e.u$ holds.

8.2. Solution

We solve this problem by using ranker S as explained below. First, recall that the mutual exclusion problem is solved using ranker C; therefore, as the resource allocation problem is a generalization of the mutual exclusion problem, the resource allocation problem is solved by a ranker at least as powerful as ranker C. Next, we argue the need for stability, and hence the need for ranker S. Assume to the contrary that we solve this problem using ranker C (i.e., a ranker without stability) as follows: A process wishing to access a copy of the resource requests and obtains a rank. After that it compares its rank with those of the other processes and accesses the resource if its rank is one of the K highest ones. Now, consider a process u that is waiting to access a copy of the resource. Because of the absence of stability, process u can make no assertions about the rank of another process if the state of the other process is *grey*. Therefore, process u has to wait until it observes the state of the other processes to be *white* or *black*. But, because up to K different processes may be accessing the resource at any given time, it is possible for all other processes to be continuously cycling through the states t, h, and e and, therefore, it is possible that process u always observes the state of other processes to be *grey* (i.e., when they are obtaining a rank). When this happens, process u waits forever, thus violating the no starvation condition. It is due to the above reason that we require stability (and hence, ranker S) to solve this problem. The above scenario does not occur if we use ranker S because if another process v continuously keeps cycling through its states while process u is waiting, then eventually process u will have a higher rank than process v (due to precedence) and will continue to have a higher rank (due to stability) until it accesses the resource. (It should be noted that the above problem due to continuously cycling of processes does not occur if a process that is in the process of comparing ranks prohibits other processes from changing their states. However, such a solution reduces concurrency and is not considered further.)

The program for process u is as follows. Predicate $high.u.v$ is defined to be $white.v \vee r.v \prec r.u$. The expression $(\#v :: checked.u.v)$ denotes the number of variables $checked.u.v$ which are *true*.

initially $white.u \wedge t.u \wedge (\forall v :: \neg checked.u.v)$
assign

$mode.u, state.u := h, grey$	**if** $t.u$
$[]\ \langle []\ v :: checked.u.v := true$	**if** $h.u. \wedge black.u \wedge high.u.v \rangle$
$[]\ mode.u := e$	**if** $h.u \wedge black.u \wedge (\#v :: checked.u.v) \geqslant N - K$
$[]\ mode.u, state.u := t, white$	**if** $e.u$
$\|\ \langle \|\ v :: checked.u.v := false$	**if** $e.u \rangle$

end

Process u manipulates variables *state.u* and *mode.u* as follows. When it sets *mode.u* to *h*, it also sets *state.u* to *grey* (to request a new rank). Then, it waits for *state.u* to become *black* (this happens eventually due to responsiveness). After that, it compares its rank with all other processes and sets boolean variable *checked.u.v* to *true* if process v is either *white* (in which case, on account of precedence, process v will have a lower rank than process u when it becomes hungry) or has a lower rank. Once at least $N-K$ of these variables become *true*, the process transits from hungry to eating state. Finally, when process u completes eating and sets *mode.u* to *t*, it also sets *state.u* to *white*, and resets each variable *checked.u.v* to *false*.

8.3. Proof of correctness

A formal proof of correctness for this solution is detailed in [37]; we present a sketch of this proof. Recall that ranker S satisfies the properties of responsiveness, precedence, acyclicity, comparability, and acyclicity. These five properties are used to prove that our solution satisfies mutual exclusion and no starvation. The proof of mutual exclusion is based on the precedence, acyclicity, comparability, and stability properties while the proof of no starvation is based on all the five properties of ranker S—responsiveness, precedence, acyclicity, comparability, and stability.

Our solution satisfies the mutual exclusion condition because of the following invariant (due to precedence and stability)—an eating process is *black*, and the number of *black* processes with a rank higher than an eating process is less than K. Now, due to the total order on the ranks (ensured by acyclicity and comparability), if more than K processes are eating at the same time, then one of them has a rank lower than K other processes. But, this contradicts the invariant and, therefore, at most K processes can eat at any one time.

To see why our solution is free from starvation, consider any process u that is hungry and let v be any other process. Because of the responsiveness and comparability properties of ranker S, eventually either the predicate *high.u.v* holds (in which case process u does not wait for process v) and continues to hold (due to stability), or process v becomes *black* and has a higher rank than u (in which case process u waits for process v). In the latter case, consider the directed *wait-for* graph rooted at u. Because of acyclicity of ranker S, this graph is acyclic and, therefore, has some leaf nodes. Eventually, each process at a leaf node will start eating and will later start thinking (because all eating periods are finite). This reduces the size of the *wait-for* graph (due to precedence). It follows by induction on the size of the *wait-for* graph that it eventually contains no edges, i.e., process u is not waiting for any other process. When this happens, process u will make a hungry to eating transition, thus ensuring no starvation.

9. Implementation of rankers

In this section we present implementations for the four rankers—Φ, A, C, and S, discussed earlier. As mentioned earlier, program *ranker* is designed to be a composition of local ranker submodules *ranker.u*, one for every process. Submodule *ranker.u* is responsible for ranking process u and it does not modify the interface variables, *state.v* and *r.v*, of any other user process. The implementations that are presented here are highly concurrent as they use a fine grain of atomicity (i.e., every statement reads or writes at most one shared variable) and are "wait-free" (i.e., the ranking of a process is done within a bounded number of steps [2,19]).

9.1. Implementation of ranker Φ

Recall that ranker Φ is required to satisfy two properties—responsiveness, a local property over processes, and precedence, a pairwise property over processes. Thus, this ranker is not required to satisfy any global properties and, consequently, this ranker composes, i.e., given two ranker implementations for m and n processes, we can compose them (after a renaming of local variables if needed) to obtain a ranker implementation for $m + n$ processes. Based on this observation, we define an implementation for two processes; the implementation for any arbitrary number of processes is obtained by repeated composition.

Let u and v be two user processes. As stated earlier, we design program *ranker* to be the composition of two submodules, *ranker.u* and *ranker.v*, which we define next. Variables *r.u* and *r.v* take on integer values and the ranking relation $\prec$ is defined as follows:

$$r.u \prec r.v \equiv r.u < r.v,$$

where $<$ is the less-than relation on integers. Because $<$ is a total order, $\prec$ is irreflexive and does not contain cycles of length two.

The design of program *ranker.v* is simple—when process v turns *grey*, variable *r.v* is assigned a value less than *r.u*, and then *state.v* is set to *black*. Observe that, if *precedes.u.v* holds and processes u and v are *black*, then variable *r.v* contains a value less than *r.u* and, consequently, $r.v \prec r.u$, thus satisfying the precedence condition. Program *ranker.u* is given below (program *ranker.v* can be obtained by a simple substitution). Variable *b.u* is a program counter, variable *t.u* stores intermediate values, and variable *checked.u.v* is a boolean and is *true* iff the intermediate value for process v has been computed.

initially $r.u = 0 \wedge b.u = 0 \wedge \neg checked.u.v$

assign

$b.u := 1$	**if** $b.u = 0 \wedge grey.u$
$[]\ t.u, checked.u.v := r.v - 1, true$	**if** $b.u = 1 \wedge \neg checked.u.v$
$[]\ r.u, b.u := t.u, 2$	**if** $b.u = 1 \wedge checked.u.v$
$[]\ state.u, b.u, checked.u.v :=$	
$\quad black, 0, false$	**if** $b.u = 2$

end

Program *ranker.u* sets $b.u$ to 1 if *state.u* is *grey*. Then, it reads the rank of process v and sets variable $t.u$ to the value $r.v - 1$. After that, the rank of process $r.u$ is set to $t.u$. Finally, the state of process u is set to *black*, and variables $b.u$ and *checked.u.v* are reset. Variable $b.u$ is used to model sequential composition of statements; in fact the above four statements are executed in a strict sequential order. Observe that we have used a very fine grain of atomicity as each statement reads or writes at most one shared variable. Also, the above program is "wait-free" as the ranking of process u is completed within four steps. This ensures that the responsiveness condition is satisfied.

When the above implementation for two processes is extended to an arbitrary number of processes, variables $r.u$ and $t.u$ become arrays with one component per process, and the definition of $\prec$ is changed to

$$r.u \prec r.v \equiv r.u.v < r.v.u,$$

where $r.u.v$ denotes the component of array $r.u$ corresponding to process v.

9.2. Implementation of ranker A

Because ranker A satisfies acyclicity, a global property over processes, this ranker does not compose. However, the implementation is similar to that for ranker Φ presented in the previous subsection. Variable $r.u$ takes on integer values and the ranking relation is defined as before:

$$r.u \prec r.v \equiv r.u < r.v.$$

When process v turns *grey*, variable $r.v$ is assigned a value less than the rank of *all other black* processes. Consequently, if predicate *precedes.u.v* holds, then variable $r.v$ is assigned a value less than $r.u$ (note that *precedes.u.v* $\Rightarrow$ *black.u*) during the ranking process. Thus, when process v turns *black*, the rank of process v is less than that of process u, as required by the precedence condition. The proof of acyclicity follows due to the acyclicity of relation $<$.

In the code for *ranker.u* presented below, variables $b.u$, $t.u.v$, and $c.u.v$ are all local variables; variables $b.u$ and $c.u.v$ are program counters and variable $t.u.v$ stores the intermediate value with respect to process v. When process u

turns *grey*, variable $b.u$ is set to 1. Then, the state of all other processes is checked; if the state of v is non-*black*, then $t.u.v$ is set to 0 and $c.u.v$ is set to 2; otherwise, $t.u.v$ is set to $r.v - 1$ in two successive steps (in order to use a fine grain of atomicity) and $c.u.v$ is set to 1 and then to 2. After all the intermediate values $t.u.v$ have been computed (i.e., $c.u.v = 2$ for all other processes), $r.u$ is set to the minimum of those values. Finally, the state of u is set to *black* and all program counters are reset. In the program below, variable v ranges over all processes distinct from u and the value $(min\ i :: x.i)$ refers to the minimum value among the $x.i$'s.

initially $r.u = 0 \wedge b.u = 0 \wedge (\forall v :: c.u.v = 0)$
assign

$b.u := 1$	**if** $b.u = 0 \wedge grey.u$
$[]\ \langle []\ v :: c.u.v, t.u.v := 2, 0$	**if** $b.u = 1 \wedge$ $c.u.v = 0 \wedge \neg black.v$
$\quad c.u.v := 1$	**if** $b.u = 1 \wedge c.u.v = 0 \wedge black.v$
$\quad c.u.v, t.u.v := 2, r.v - 1$	**if** $c.u.v = 1$
$\rangle$	
$[]\ b.u, r.u := 2, (min\ v :: t.u.v)$	**if** $b.u = 1 \wedge (\forall v :: c.u.v = 2)$
$[]\ b.u, state.u := 0, black$	**if** $b.u = 2$
$\|\ \langle \|\ v :: c.u.v := 0$	**if** $b.u = 2\rangle$

end

Once more, we have used a very fine grain of atomicity (as each statement reads or writes at most one shared variable), and program *ranker.u* is "wait-free" (as the ranking of a process is completed within $3n$ steps where n is the number of processes). The proofs of responsiveness and precedence are similar to the proofs in the previous implementation and as stated earlier, the proof of acyclicity follows from the acyclicity of the relation $<$ on integers.

9.3. Implementation of ranker C

The programs for this implementation are as in the previous implementation; only the ranking relation $\prec$ is defined differently. It is now defined to be the lexicographic ordering of the previous ranking relation and some fixed total order q, i.e., for all distinct u and v,

$$r.u \prec r.v \quad \equiv \quad r.u < r.v \ \vee \ (r.u = r.v \wedge q.u.v).$$

The proofs of responsiveness and precedence are as in the previous implementations. Properties acyclicity and comparability follow from the total order of $\prec$. This construction does not guarantee stability as variables $r.u$ are not monotonically decreasing. As an example, consider the following history of

processes u and v with $q.u.v$ as false. In the initial state, process u requests and obtains a rank and becomes *black* with $r.u = 0$. Then, process v requests and obtains a rank, and becomes *black* with $r.v = -1$. Processes u and v now turn *white*. Then, processes u and v request ranks concurrently by turning *grey* and observe the state of the other process to be *grey*. Process u thus sets $r.u$ to 0 and turns *black*. In this state the antecedent of the stability property, $black.u \wedge r.v \prec r.u$, holds. Finally, process v, having observed the state of process u to be *grey*, sets $r.v$ to 0. This means that in the resulting state $r.v \prec r.u$ no longer holds (as $r.u = r.v$ and $q.u.v$ is false). Consequently, the property of stability is violated.

9.4. Implementation of ranker S

This implementation is similar to the previous implementation. The ranking relation and the program variables are exactly the same; the only difference is that variable $r.u$ is now ensured to be monotonically decreasing (in order to satisfy stability). The program is as follows.

initially $r.u = 0 \wedge b.u = 0 \wedge (\forall v :: c.u.v = 0)$
assign

$b.u := 1$	**if** $b.u = 0 \wedge grey.u$
$[] \langle [] v :: c.u.v, t.u.v := 2, 0$	**if** $b.u = 1 \wedge$
	$\quad c.u.v = 0 \wedge \neg black.v$
$\quad c.u.v := 1$	**if** $b.u = 1 \wedge c.u.v = 0 \wedge black.v$
$\quad c.u.v, t.u.v := 2, r.v - 1$	**if** $c.u.v = 1$
$\rangle$	
$[] b.u, r.u :=$	
$\quad 2, min(r.u, (min\ v :: t.u.v))$	**if** $b.u = 1 \wedge (\forall v :: c.u.v = 2)$
$[] b.u, state.u := 0, black$	**if** $b.u = 2$
$\| \langle \| v :: c.u.v := 0$	**if** $b.u = 2 \rangle$

end

The proofs of responsiveness, precedence, acyclicity, and comparability are as in the previous implementation. The proof of stability follows from the fact that each variable $r.u$ is monotonically decreasing.

10. Concluding remarks

Our aim in this paper is to study synchronization problems by defining the interface between ranker implementations and ranker applications. For

this purpose, we identified five properties of rankers—responsiveness, precedence, acyclicity, comparability, and stability—and obtained a hierarchy of four rankers based on these properties. We have showed that such a separation of concerns leads to simple and modular solutions to various synchronization problems.

The first evidence of modularity comes from our ability to solve all synchronization problems by the same strategy, thus yielding similar solutions to seemingly different problems. As stated earlier, all our solutions consist of three steps. First, a process that wishes to perform a critical action sets its state to *grey* and waits for a rank. Then, on obtaining a rank (signaled by the state of the process becoming *black*) the process compares its rank with those of competing processes and proceeds accordingly. Finally, when the process completes its critical action, it informs the ranker by setting its state to *white*. We used the above strategy to solve a number of problems in Sections 6, 7, and 8.

Another evidence of modularity comes from the fact that small modifications in the problem description do not lead to major modifications in the solution. For example, when we add the timeout requirement to the dining philosophers problem (Section 7.4), our solution of the new problem is very similar to that of the original problem; in fact, the only difference between the two solutions is that now we use ranker S instead of ranker C, and a simple disjunct is added to the condition under which a philosopher makes a transition to the thinking state.

Finally, the modularity of our solutions is also reflected in the modularity of the proofs of correctness; solutions to similar problems share a considerable amount of proof effort. For example, the proofs of mutual exclusion for the dining philosophers problem without timeouts and for the dining philosophers problem with timeouts are almost the same; only one lemma has to be reproved [37]. The same assertion can also be made about the rest of the proofs. In summary, the abstraction of rankers allows us to design simple modular solutions to synchronization problems.

As stated earlier, we chose the five properties of rankers by examining the synchronization problems in the literature. Our main concerns in choosing the properties were simplicity and completeness. Each one of the five properties addresses an orthogonal aspect of synchronization and is motivated by a particular class of synchronization problems (for example, comparability is motivated by problems requiring a total order, and stability is motivated by timeouts). We elaborate on the formulations of responsiveness and precedence next.

Earlier we stated the requirement of responsiveness as $grey.u \mapsto black.u$, for each process u. This states that a process that wishes to obtain a rank eventually does so. As a result, it may appear that the computation of ranks can be done in a critical section by using a starvation-free mutual exclusion

algorithm. However, that is not what we have in mind. We require that the computation of ranks be done in a wait-free manner, i.e., within a bounded number of steps irrespective of the execution of the other processes. The requirement of wait-freedom is difficult to state in UNITY. As suggested by Lamport [28], one way to state it may be to divide the ranker module into submodules *ranker.u*, one for every process, and require that rank and the state of process u be local to *ranker.u*. Then we require that no matter how the other submodules are implemented, the implementation of *ranker.u* should ensure that the composition of all the submodules meets the progress property $grey.u \mapsto black.u$. Such a formalization ensures that submodule *ranker.u* does not rely on other non-local submodules in achieving the state transition from *grey* to *black*. Formally, this would be stated as a conditional property in UNITY.

The statement of precedence may seem to be unnecessarily complex. This complexity is due to the fact that it is impossible to implement a ranker that orders the ranks of the processes based on the order in which they turn *grey* (i.e., ask for ranks) or turn *black* (i.e., receive ranks). These impossibility results are discussed next. Suppose we require a process to have a higher rank if it turned *grey* before another process, i.e., if process u turned *grey* before process v then process u has a higher rank. Such a ranker cannot be implemented because the state transitions from *white* to *grey* occur in the user processes asynchronously with the ranker and, therefore, it is not possible for a ranker implementation to detect every occurrence of a state in which one process is *grey* and the other is *white*. A similar result was proved by Lamport in [25].

Consider a different formulation of precedence. Suppose we require that a process has a higher rank if it turned *black* before another process. In other words, if process u turned *black* while another process v is not *black*, then process u has a higher rank. Also suppose that we require the ranker programs to be wait-free and use a fine grain of atomicity. Such a ranker cannot be implemented because it can be used to solve the binary election problem which has been shown to be impossible in [2]. The proof by reduction appears in [37]. The above two impossibility results led us to the current formulation of precedence in which the state interval over which a process is *grey* is the basis for ordering ranks.

Acknowledgement

We are grateful to Leslie Lamport for his comments regarding the wait-freedom requirements in the specification of rankers and the first-come, first-served doorway. We are also grateful to Edsger W. Dijkstra and Jayadev Misra for reading earlier versions of this paper.

References

[1] D. Agrawal and A. El Abbadi, An efficient solution to the mutual exclusion problem, in: *Proceedings Eighth Annual ACM Symposium on the Principles of Distributed Computing* (1989) 193–200.

[2] J.H. Anderson and M.G. Gouda, The virtue of patience—concurrent programming with and without waiting, Tech. Report TR.90.23, Department of Computer Sciences, University of Texas at Austin (1990).

[3] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg and R. Reischuk, Renaming in an asynchronous environment, *J. ACM* **37** (3) (1990) 524–548.

[4] B. Bloom, Constructing two-writer atomic registers, in: *Proceedings Sixth Annual ACM Symposium on Principles of Distributed Computing* (1987) 249–259.

[5] P. Brinch Hansen, Concurrent programming concepts, *ACM Comput. Surveys* **5** (1973) 223–245.

[6] G.N. Buckley and A. Silberschatz, An effective implementation for the generalized input–output construct of CSP, *ACM Trans. Programming Languages Syst.* **5** (2) (1982) 223–235.

[7] J.E. Burns and G.L. Peterson, Constructing multi-reader atomic values from non-atomic values, in: *Proceedings Sixth Annual ACM Symposium on Principles of Distributed Computing* (1987) 222–231.

[8] K.M. Chandy, A mutual exclusion algorithm for distributed systems, Tech. Report, Department of Computer Sciences, The University of Texas (1982).

[9] K.M. Chandy and J. Misra, The drinking philosophers problem, *ACM Trans. Programming Languages Syst.* **6** (4) (1984) 144–156.

[10] K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation* (Addison-Wesley, Reading, MA, 1988).

[11] P.J. Courtois, F. Heymans and D.L. Parnas, Concurrent control with 'readers' and 'writers', *Comm. ACM* **14** (10) (1971) 667–668.

[12] E.W. Dijkstra, Cooperating sequential processes, Tech. Report EWD-123, Technological University, Eindhoven, Netherlands (1965).

[13] E.W. Dijkstra, Solution of a problem in concurrent program control, *Comm. ACM* **8** (9) (1965) 320–322.

[14] E.W. Dijkstra, Hierarchical ordering of sequential processes, in: C.A.R. Hoare and P.J. Perrott, eds., *Operating Systems Techniques*, (Academic Press, London, 1972) 72–93.

[15] D. Dolev and N. Shavit, Bounded concurrent time-stamp systems are constructible, in: *Proceedings Twenty-first Annual ACM Symposium on the Theory of Computing* (1989) 454–465.

[16] C. Dwork and O. Waarts, Simple and efficient bounded concurrent timestamping, in: *Proceedings Twenty-fourth Annual ACM Symposium on the Theory of Computing* (1992) 655–666.

[17] K.P. Eswaran, J.N. Gray, R.A. Lorie and I.L. Traiger, The notions of consistency and predicate locks in database systems, *Comm. ACM* **19** (11) (1976) 624–633.

[18] M.J. Fischer, N. Lynch, J.E. Burns and A. Borodin, Distributed FIFO allocation of identical resources using small shared variables, *ACM Trans. Programming Languages Syst.* **11** (1) (1989) 90–118.

[19] M. Herlihy, Wait-free implementations of concurrent objects, in: *Proceedings Sixth ACM Symposium on Principles of Distributed Computing* (1988) 276–290.

[20] C.A.R. Hoare, An axiomatic basis for computer programming, *Comm. ACM* **12** (1969) 576–580.

[21] A. Israeli and M. Li, Bounded time-stamps, in: *Proceedings Twenty-eighth Annual IEEE Symposium on Foundations of Computer Science*, Los Angeles, CA (1987) 371–382.

[22] L. Lamport, A new solution of Dijkstra's concurrent programming problem, *Comm. ACM* **17** (8) (1974) 453–455.

[23] L. Lamport, The synchronization of independent processes, *Acta Inform.* **7** (1) (1976) 15–34.

[24] L. Lamport, Time, clock, and the ordering of events in a distributed system, *Comm. ACM* **21** (7) (1978) 558–565.

[25] L. Lamport, What it means for a concurrent program to satisfy a specification: why no one has specified priority, in: *Proceedings Eleventh Annual ACM Symposium on the Principles of Programming Languages*, Salt Lake City, UT (1984) 78–83.
[26] L. Lamport, The mutual exclusion problem, Part II: Statement and solutions, *J. ACM* **33** (2) (1986) 327–348.
[27] L. Lamport, On interprocess communication, Parts I and II, *Distributed Comput.* **1** (1986) 77–101.
[28] L. Lamport, Private communication.
[29] J. Misra, Specifying concurrent objects as communicating processes, *Sci. Comput. Programming* **14** (1990) 159–184.
[30] R. Newman A protocol for wait-free, atomic, multi-reader shared variables, in: *Proceedings Sixth Annual ACM Symposium on Principles of Distributed Computing* (1987) 232-249.
[31] G.L. Peterson, Myths about the mutual exclusion problem, *Inform. Process. Lett.* **12** (3) (1981) 115–116.
[32] G.L. Peterson and J.E. Burns, Concurrent reading while writing II: the multiwriter case, in: *Proceedings Twenty-eighth Annual IEEE Symposium on Foundations of Computer Science*, Los Angeles, CA (1987) 383–392.
[33] G.L. Peterson and M.J. Fischer, Economical solutions for the critical section problem in a distributed system, in: *Proceedings Ninth Annual ACM Symposium on Theory of Computing* (1977) 91–97.
[34] K. Raymond, A tree-based algorithm for distributed mutual exclusion, *ACM Trans. Comput. Syst.* 7 (1) (1989) 61–77.
[35] D.P. Reed and R.K. Kanodia, Synchronization with eventcounts and sequencers, *Comm. ACM* **22** (2) (1979) 115–123.
[36] G. Ricart and A. Agrawala, An optimal algorithm for mutual exclusion in computer networks, *Comm. ACM* **24** (1) (1981) 9–17.
[37] A.K. Singh, Ranking in distributed systems, Ph.D. Dissertation, The University of Texas at Austin, Austin, TX (1989).
[38] A.K. Singh, J.H. Anderson and M.G. Gouda, The elusive atomic register revisited, in: *Proceedings Sixth ACM symposium on Principles of Distributed Computing* (1987) 206–221.